# Assembly Cookbook for the Apple™ II/IIe

part one

Don Lancaster

# Assembly Cookbook for the Apple™ II/IIe

# ( part one )

by

Don Lancaster

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com
ISBN: 978-1-882193-16-5

THIRD EDITION
FIRST PRINTING 2011



International Standard Book Number: 978-1-882193-16-5

# Contents

# Why you gotta learn Machine Language

Check into **Softalk** magazine's listing of the "top thirty" programs for your Apple. II or lie, and you'll find that thirty out of thirty of this month's winners usually involve mac:hine language programs or support modules, written by authors who use assemblers and who make use of assembly language programming skills.

And, last month's top thirty were also swept by machine language, thirty to zip. And next month's listings probably will be the same. Somehow, thirty to zero seems statistically significant. There's got to be a message there.

Yep.

So, on the basis of what is now happening in the real world, you can easily conclude that…

> **The only little thing wrong with BASIC or Pascal is that it is categorically impossible to write a decent Apple II or lie program with either of them!**

Naturally, things get even worse if you try to work in some specialty language, such as FORTH, PILOT, LOGO, or whatever, since you now have an even smaller user and interest base and thus an even more miniscule market.

What would happen if, through fancy packaging, heavy promotion,

or outright lies, a BASIC or a Pascal program somehow happened to blunder into the top thirty some month?

One of three things…

**— maybe —**

1. Word will quickly get out over the bulletin board systems and club grapevines over how gross a ripoff the program is, and the the program will ignominiously bomb out of sight.

**— or —**

2. A competitor will recognize a germ or two of an undeveloped idea in the program and come up with a winning machine language replacement that does much more much faster and much better, thus running away with all the marbles.

**— or, hopefully —**

3. The program author will see the blatant stupidity of his ways and will rework the program into a decent, useful, and popular machine language version.

The marketplace has spoken, and its message is overwhelming…

**If you want to write a best-selling or any money-making program for the Apple II or IIe, the program must run in machine language.**

OK, so it's obvious that all the winning Apple II programs run in machine language. But, why is this so? What makes machine language so great? How does machine language differ from the so-called "higher level" languages? What is machine language all about?

Here are a few of the more obvious advantages of machine language…

**MACHINE LANGUAGE IS —**

**Fast**
**Compact**
**Innovative**
**Economical**

**Flexible**
**Secure**
**User Friendly**
**Challenging**

**Profitable**

That's a pretty long list and a lot of heavy claims. Let's look at a few of the big advantages of machine language one by one…

## Speed

It takes from two to six millionths of a second, or microseconds, to store some value using Apple's 6502 machine language. Switch to interpreted Integer BASIC or Applesoft, and similar tasks take as much as two to six thousandths of a second, or milliseconds. This is slower by a factor of one thousand.

The reason for the 1000:1 speed difference between interpreted "high level" languages and machine language is that there are bunches of housekeeping and overhead involved in deciding which tasks have to be done in what order, and in keeping things as programmer friendly as possible.

Now, at first glance, speed doesn't seem like too big a deal. But speed is crucial in many programs. Let's look at three examples.

For instance, a word processor program that inserts characters slower than you can type is a total disaster, for one or more characters can get dropped. Even if it doesn't drop characters, a word processor that gets behind displaying stuff on the screen gets to be very infuriating and annoying. So, word processing is one area where machine language programs are an absolute must, because of the needed speed.

Business sorts and searches are another area where the speed of machine language makes a dramatic difference. Several thousand items sorted in interpreted BASIC using a bubble sort might take a few hours. Go to a quicksort under machine language, and the same job takes a few seconds at most. Thus, any business program that involves sorts and searches of any type is a prime candidate for machine language.

Finally, there is any program that uses animation. Interpreted BASIC is way too slow and far too clumsy to do anything useful in the way of screen motion, game responses, video art, and stuff like this. Thus, all challenging or interesting games need machine language to keep them that way.

But, you may ask, what about compilers? Aren't there a bunch of very expensive programs available that will compile BASIC listings into fast-running machine language programs? Sure there are.

Most compiled code usually runs faster than interpreted code. But, when you find the real-world speedup you get and compare it to the same program done in machine language by a skilled author, it is still no contest…

**Most programs compiled from a "higher level" language will run far slower, and will perform far more poorly, than the same task done in a machine language program written by a knowing author.**

Some specifics. If you run exactly the worst-case benchmark program on one of today's highly promoted compiler programs, you get a blinding speedup of 8 percent, compared to just using plain old interpreted BASIC. Which means that a task that took two hours and fifty five minutes can now be whipped through in a mere two hours and forty-two minutes instead.

Golly gee, Mr. Science. Actually, most compilers available today will in fact speed up interpreted programs by a factor of two to five. This is certainly a noticeable difference and is certainly a very useful speedup. But it is nothing compared to what experienced machine language authors can do when they attack the same task.

A compiler program has to make certain assumptions so that it can work with all possible types of program input. Machine language authors, on the other hand, are free to optimize their one program to do whatever has to be done, as fast, as conveniently, and as compactly as possible. This is the reason why you can always beat compiled code if you are at all into machine.

Another severe limitation of Applesoft compilers is that they still end up using Applesoft subroutines. These subroutines may be just plain wrong (such as RND), or else may be excruciatingly slow (such as HPLOT). Hassles like these are easily gotten around by direct machine programming.

Some machine language programs are faster than others. Most often, you end up trading off speed against code length, programming time, and performance features.

One way to maximize speed of a machine language program is to use brute-force coding, in which every instruction does its thing in the minimum possible time, using the fastest possible addressing modes. Another speed trick is called table lookup, where you look up an answer in a table, rather than calculating it. One place where table lookup dramatically speeds things up is in the Apple II HIRES graphics routines where you are trying to find the address of a display line. Similar table lookups very much quicken trig calculations, multiplications, and stuff like this.

So, our first big advantage of machine language is that it is ridiculously faster than an interpreted high level language, and much faster than a compiled high level language.

### Size

A controller program for a dumb traffic light can be written in machine language using only a few dozen bytes of code. The same thing done with BASIC statements takes a few hundred bytes of code, not counting the few thousand bytes of code needed for the BASIC interpreter. So, machine language programs often can take up far less memory space than BASIC programs do.

Now, saving a few bytes of code out of a 64K or 128K address space may seem like no big deal. And, it is often very poor practice to spend lots of time to save a few bytes of code, particularly if the code gets sneaky or hard to understand in the process.

But, save a few dozen bytes, and you can add fancy sound to your program. Save a few thousand bytes more, and you can add HIRES graphics or even speech. Any time you can shorten code, you can make room for more performance and more features, by using up the new space you created. Save bunches of code, and you can now do stuff on a micro that the dino people would swear was impossible.

Three of the many ways machine language programs can shorten code include using loops that use the same code over and over again, using subroutines that let the same code be reached from a few different places in

a program, and using reentrant code that calls itself as often as needed. While these code shortening ideas are also usable in BASIC, the space saving results are often much more impressive when done in machine language.

Machine language programs also let you put your files and any other data that go with the program into its most compact form. For instance, eight different flags can be stuffed into a single code word in machine language, while BASIC normally would need several bytes for each individual flag.

Which brings us to another nasty habit compilers have.

Compilers almost always make an interpreted BASIC program longer so that the supposedly "faster" compiled code takes up even more room in memory than the interpreted version did. The reason for this is that the compiler must take each BASIC statement at face value, when and as it comes up. The compiler then must exactly follow the form and structure of the original interpreted BASIC code. Thus, what starts out as unnecessarily long interpreted code gets even longer when you compile it.

Not to mention the additional interpretive code and run time package that is also usually needed.

A machine language programmer, on the other hand, does not have to take each and every BASIC statement as it comes up. Instead, he will write a totally new machine language program that, given the same inputs, provides the same or better outputs than the BASIC program did. This is done by making the new machine language program have the same function that the BASIC one did, but completely ignoring the dumb structure that seems to come with the BASIC territory.

The net result of all this is that a creative machine language programmer can often take most BASIC programs and rewrite them so they are actually shorter. As a typical example, compare your so-so adventure written in BASIC against the mind blowers written in machine. When it comes to long files, elaborate responses, and big data bases, there is no way that BASIC can compete with a machine language program, either for size or speed.

Let's check into another file-shortening example, to see other ways that machine language can shorten code. The usual way a higher level language handles words and messages is in ASCII code. But studies have shown that ASCII code is only 25 percent efficient in storing most English text. Which means that you can, in theory, stuff four times as many words or statements into your Apple as you thought you could with ASCII.

You do this by using some text compaction scheme that uses nonstandard code manipulated by machine language instructions. For instance, in **Zork**, three ASCII characters are stuffed into two bytes of code. This gives you an extra 50 percent of room on your diskettes or in your Apple. In the **Collossial Cave** adventure version by **Adventure International**, unique codings are set aside for pairs of letters, giving you up to 100 percent more text in the same space. This means that this entire classic adventure text now fits inside the Apple, without needing any repeated disk access.

Dictionary programs use similar compaction stunts to minimize code length. If the words are all in alphabetical order, you can play another compaction game by starting with a number that tells you how many of the beginning letters should stay the same in the next word, and by using

another coding scheme to add standard endings (-s, -ing, -ed, -ly, etc…) to the previous word.

The bottom line is that machine language programs can shorten code enough that you can add many new features to an existing program, can put more information in the machine at once, or can cram more data onto a single diskette.

### Innovation—Finding the Limits

One really big advantage to machine language on the Apple II or IIe is that it pushes the limits of the machine to the wall. We now can do things that seemed impossible only a short while ago. This is done by discovering new, obscure, and mind-blowing ways to handle features using machine language code.

Some ferinstances…

With BASIC, you can get only one obnoxious beep out of the Apple's on-board speaker. Play around with PEEKs and POKEs, and you can get a few more pleasant buzzes and low-frequency notes. This is almost enough to change a fifth rate program into a fourth rate one.

Now, add a short machine language program, and you can play any tone of any duration. But, that's old hat. The big thing today is known as **duty cycling**. With duty cycling done from a fairly fancy machine language driver, you can easily sound the on-board speaker at variable volume, with several notes at once, or even do speech with surprisingly good quality.

All this through the magic of machine language, written by an author who uses assemblers and who posesses at least a few assembly language programming skills.

The Apple II colors are another example. The HIRES subs in BASIC only give you 16 LORES colors and a paltry 6 HIRES colors. But, go to machine language, and you end up with at least 121 LORES colors and at least 191 HIRES ones on older Apples. The Apple Iie offers countless more.

And that's today. Even more colors are likely when the machine language freaks really get into action.

Another place where limits are pushed by machine language is in animation and HIRES plotting. You can clear the HIRES screen seven times faster than was thought possible, by going to innovative code. You can plot screen locations much faster today through the magic of table lookup and brute-force coding. Classic cell animation is even possible.

Disk drive innovations are yet another example. Change the code and you can load and dump diskettes several times faster than you could before. You can also store HIRES and LORES pictures in many fewer sectors than was previously thought possible. Again, it is all done by creative use of machine language programs that are pushing the limits of the Apple.

A largely unexplored area of the Apple II involves exact field sync, where an exact and jitter-free lock is done to the screen. This lets you mix and match text, LORES, and HIRES on the screen, do gray scale, precision light pens, gentle scrolls, touch screens, flawless animation, and much more.

All this before the magic of all the new cucumber cool 65C02 chips,

which can allow a mind-boggling animation of **fifty times** compared to what the best of today's machine language programmers are using. But that's another story for another time.

And, exciting as the pushed limits are, we are nowhere near the ultimate»

**Today's machine language programs are nowhere near pushing the known limits of the Apple II's hardware**

And, of course…

**The known limits of the Apple II hardware are nowhere near the real limits of the Apple II hardware.**

What haven't we fully explored with the Apple II yet? How about gray scale? Anti-aliasing? Three-D graphics displays? "Picture processing" for plotters that is just as fast and convenient as "word processing"? Using the Apple as an oscilloscope? A voltmeter? Multi-Apple games, where each combatant works his own machine in real time? Scan length coded video? That SOX animation speedup? Networking?

And the list goes on for thousands more. If it can be done at all, chances are an Apple can help you do it, one way or another.

### Getting Rid of Fancy Hardware

Machine language is often fast enough and versatile enough to let you get rid of fancy add-on hardware, or else let you dramatically simplify and reduce needed hardware. This is why machine language is economical.

For instance, without machine language drivers on older Apples, you are stuck either with a 40-character screen line, or else have to go to a very expensive 80-column card board. But with the right drivers, you can display 40, 70, 80 or even 120 characters on the screen of an unmodified Apple II with no plug-in hardware. This is done by going to the HIRES screen and by using more compact fonts. You can also have many different fonts this way, upper or lower case, in any size and any language you like.

As a second example of saving big bucks with machine language, one usual way to control the world with an Apple II involves a BSR controller plug-in card, again full of expensive hardware. But you can replace all this fancy hardware with nothing but some machine language code and a cheap, old, ultrasonic burglar alarm transducer.

As yet a final example, by going to the Vaporlock exact field sync, machine language software can replace all the custom counters eeded for a precision light pen or for a touch screen. With zero hardware modifications.

In each of these examples, the machine language code is fast enough that it can directly synthesize what used to be done with fancy add-on hardware.

So, our fourth big plus of machine language is that it can eliminate, minimize, or otherwise improve add-on hardware at very low cost.

### Other Advantages

Those are the big four advantages of machine language. Speed, program size, innovation, and economy. Let's look at.some more advantages…

Machine language code is very flexible. Have you ever seen Kliban's cartoon "Anything goes in Heaven," where a bunch of people are floating around on clouds doing things that range from just plain weird to downright obscene?

Well, anything goes in machine language as well. Put the program any place you want to. Make it as long or as short as you want. "What do you mean I can't input commas?" Input what you like, when you like, how you like. Change the program anyway you want to, anytime you want to. That's what flexibility is all about.

Machine language offers solace for the security freak.

I'm not very much into program protection myself, since all my programs are unlocked, include full source code, and are fully documented. I, like practically every other advanced Apple freak, fiendishly enjoy tearing apart all "protected" programs the instant they become available, because of the great sport, humor, learning, and entertainment value that the copy protection mafia freely gives us.

And surprise, surprise. Check the **Softalk** score sheets, and you'll find that unlocked programs are consistently outselling locked ones, and are steadily moving up in the ratings and in total sales. Which means that an un-displeased and un-inconvenienced buyer in the hand is worth two bootleg copies in the bush, any day.

Time spent "protecting" software is time blown. Why not put the effort into improving documentation, adding new features, becoming more user friendly, or doing more thorough testing instead?

But, anyway, if you are naive enough or arrogant enough to want to protect your program, there are lots of opportunities for you to do so in machine language. For openers, probably 98 percent of today's Apple II owners do not know how to open and view a machine language program. Not only are you free to bury your initials somewhere in the code, but you could hide a seven-generation genealogical pedigree inside as well. How's that for proof of ownership? And, the very nature of creative machine language programming that aims to maximize speed and minimize memory space, tends to "encrypt" your program. Nuff said on this.

Machine language programs can be made very user friendly. Most higher level languages have been designed from the ground up to be designer friendly instead. BASIC goes out of its way to be easy to learn and easy to program. So, BASIC puts the programmer first and the user last. Instead of making things as easy to program as possible, you are free in machine language to think much more about the ultimate user, and make things as convenient and comfortable as possible for the final user.

Machine language programming is challenging. Is it ever.

When you become an Apple II machine language programmer, you join an elite group of the doers and shakers of Appledom. The doing doggers. This is where the challenge is, and where you'll find all the action.

And all the nickels.

Finally, there is the bottom line advantage, the sum total of all the others. Because machine language programming is fast, compact, innovative, economical, flexible, secure, and challenging, it is also profitable. Machine language is, as we've seen, the only way to grab the brass ring and go with a winning Apple II or lie program.

Should you want to see more examples of innovative use of Apple II and IIe machine language programs, check into the **Enhancing Your Apple II** series (Sams 21822). And for down-to-earth details on forming your own computer venture, get a copy of **The Incredible Secret Money Machine**.

But, surely there must be some big disadvantages to machine language programming. If machine is so great, why don't all the rest of the languages just dry up and blow away?

Well, there is also…

## The Dark Side of Machine Language

Here's the bad stuff about machine language…

**MACHINE LANGUAGE IS ALSO —**

**Unportable**
**Tedious**
**Designer Unfriendly**
**Multi-Level**
**Hard to Change**
**Hard to Teach**
**Unforgiving**
**Ego Dependent**

What a long list. A machine language program is not portable in that it will only run on one brand of machine, and then only if the one machine happens to be in exactly the right operating mode with exactly the right add-ons for that program. This means you don't just take an Apple II machine language program and stuff it into another computer and have it work properly on first try.

Should you want to run on a different machine, you have to go to a lot of trouble to rewrite the program. Things get even messier when you cross microprocessor family boundaries. For instance, translating an Apple II program to run on an Atari at least still uses 6502 machine language coding. All you have to do is modify the program to meet all the new locations and all the different use rules. But when you go from the 6502 to a different microprocessor, and all the addressing modes and op codes will change.

A lot of people think this is bad. I don't. If you completely and totally optimize a program to run on a certain machine, then that program absolutely has to perform better than any old orphan something wandering around from machine to machine looking for a home.

Machine language involves a lot of tedious dogwork. No doubt about it. Where so-called "higher level" languages go out of their way to be easy to program and easy to use machine language does not.

There are, fortunately, many design aids available that make machine language programming faster, easier, and more convenient. Foremost of these is a good assembler, and that is what the rest of this book is all about.

Machine language is very designer unfriendly. It does not hold your hand. A minimum of three years of effort is needed to get to the point where you can see what commitment you really have to make to become a really great machine language programmer.

Machine language needs multilevel skills. The average machine language program consists of three kinds of code. These are the **elemental subroutines** that do all the gut work, the **working code** that manages the elemental subs, and finally, the **high level supervisory code** that holds everything together. In a "higher level" language, the interpreter or compiler handles all the elemental subs and much of the working code for you, "free" of charge. Different skills and different thought processes are involved in working at these three levels.

This disadvantage is certainly worth shouting over…

**THERE IS NO SUCH THING AS A "SMALL" CHANGE IN A MACHINE LANGUAGE PROGRAM!**

Thus, any change at all in a machine language program is likely to cause all sorts of new problems. You don't simply tack on new features as you need them, or stuff in any old code any old place. This just isn't done.

Actually, shoving any old code any old place is done all the time, by just about everybody. It just doesn't work, that's all.

Machine language programming is something that must be learned. There is no way for someone else to "teach" you machine language programming. Further, the skills in becoming a good machine language programmer tend to make you a lousy teacher, and vice versa.

Machine language is unforgiving, in that any change in any byte in the program, or any change in starting point, or any change of user configuration, will bomb the program and plow the works.

Some people claim that machine language code is hard to maintain. But it is equally easy to write a Pascal program that is totally unfixable and undecipherable as it is to write a cleanly self-documenting machine language program. The crucial difference is that machine language gently urges you to think about maintainability, while Pascal shoves this down your throat. Sideways.

Finally, machine language is highly ego-dependent. Your personality determines the type and quality of machine language programs you write. Many people do not have, and never will get, the discipline and sense of order needed to write decent machine language programs. So, machine language programming is not for everyone.

It is only for those few of you who genuinely want to profit from and enjoy your Apple II or IIe. That's a pretty long list of disadvantages, and it should be enough to scare mostsane individuals away from machine language. Except for this little fact…

**NONE of the disadvantages of machine language matter in the least, because there is NO OTHER ALTERNATIVE to machine language when it comes to writing winning programs for your Apple II or IIe.**

Or to rework the tired joke about the guy who slaved away all his life in Florida and then retired to New Jersey…

**Have you ever seen a machine language program that was improved by rewriting it in BASIC or Pascal?**

Not that it won't happen. It just isn't very likely, that's all.

## Getting Started

Here is how I would have you become a decent machine language programmer. First, you should write, hand-code, test, and debug several hundred lines directly in machine language, without the use of any assembler at all. The reason for this is that…

**Before you can learn to program in assembly language, you must learn how to program in machine language!**

So many assembler books and courses omit this essential first step! An assembler is simply too powerful a tool to start off with. You must first know what op codes are and how they are used. You must thoroughly understand addressing modes and the different ways they are used.

There is a series of nine discovery modules found in Volume II of **Don Lancaster's Micro Cookbook** (Sams 21829) that will take you step by step through most of the op codes of the Apple's 6502 microprocessor.

After you have done your homework and can tell the difference between a page zero and an immediate addressing command, and after you know whether "page zero, indexed by Y," is a legal command on the Apple, then, and only then, should you move up to a miniassembler, such as the excellent one in Apple's new **BUGBYTER**.

Then you run another few hundred lines of code through a miniassembler to understand what an assembler can do for you. Finally, you go on to a full blown assembler and learn assembly language programming.

The few tedious "front-end" hours spent doing everything "the hard way" will be more than made up in the speed and convenience with which you pick up assembler skills later.

This starting by hand, going to a miniassembler, and only then stepping

up to a full assembler is the way I would have you become a winning assembly language programmer.

Most people, though, will probably try…

**SNEAKING UP ON REALITY—**

**For a SIXTH rate program—**
**Write it in Pascal.**

**For a FIFTH rate program—**
**Write it in BASIC.**

**For a FOURTH rate program—**
**Write it in BASIC, but use a few PEEKs and POKEs.**

**For a THIRD rate program—**
**Write it in BASIC, but use the CALL command to link a few short machine language code segments.**

**For a SECOND rate program—**
**Write it in BASIC, but use the"&" command to link several longer blocks of machine language code.**

**For a FIRST rate program—**
**Do the whole thing in machine language like you should have done in the first place.**

The trouble with the "sneaking-up" method is that it takes you forever to see how bad "higher level" languages really are, and you spend all your time goofing around with second-rate code. But, the "sneaking-up" method does eliminate some of the cultural shock of starting straight into machine language programming from scratch.

Instead, start with and stay in machine language.

## A Plan

This book is intended to show you what an assembler is and how to use one to write profitable and truly great Apple II or lie machine language programs. You will find the book in two halves. The first half is the "theory" part that tells us all about what assemblers are and how to use them. The second half is the "practice" part that will lead you step by step through some practical ripoff modules of working assembly language code. Code that is unlocked, ready to go, wide open, and easily adapted to your own uses.

We start in chapter one by finding out what an assembler is and what it does. We then check into the popular assemblers available today, along with a list of the essential tools for assembler programming, some magazines, and other resources.

Our examples will use Apple's own newly overhauled and upgraded **EDASM** macroassembler, first because it is the one I use,

and secondly because it is the de-facto standard for assembling Apple II machine language programs. Many of the weakest features of EDASM get eliminated in one swell foop simply by using Apple Writer IIe instead of the original EDASM editor, and using the magic of WPL to help along your macros.

At any rate, most of what happens here will apply to any assembler of your choosing. We will provide source code on the companion diskette for either EDASM or the S-C Assembler formats. Just be sure to tell us which one you want. Either of these versions should be translatable to the assembler of your choice.

Most Apple-based assemblers come in two parts. One part puts together the story of what is to be done, while the second part takes the story and converts it into working machine language code. Putting together the story is called editing, while creating the machine language code is called assembling. The story or script is more properly called the **source code**, while the final program or module is usually called the **object code**.

Source code details are covered in chapter two, where we look into source code lines, fields, labels, op codes, operands, and comments, finding out just what all these are and how they are used. The structure of your source code is outlined in chapter three, where we find the 16 essential parts to an assembly language program, and how to use them. We also find out here exactly why structure of any kind is inherently evil and why structure must be avoided at all costs.

Today there are two good ways to write source code. The "old way" uses the editor provided in the assembler package. We'll cover the old way in depth in chapter four.

The "new way" uses the power of a modern word processor to do your source code entry and editing, and has bunches of potent advantages. Not the least of which is that creating and editing source code is lots more fun with a word processor, and that you can instantly upgrade a lousy editor/assembler into a super-powerful one. Drag a supervisory language such as WPL along for the ride, and you can do incredible macro-style things that otherwise would be unavailable. Chapter five tells all.

At long last, in chapter six, we get around to actually assembling source code into working object code. Here we also check into error messages, debugging techniques, and things like this. And that just about rounds out the theory half of the book.

The practice half includes nine ready-to-go **ripoff modules** that show you examples of some of the really essential stuff that's involved in Apple programming. I've tried to concentrate on the things that are really needed and really get used, such as a decent random number generator, a state-of-the-art string imbedder, an option picker, a time delay animator, two approaches to sound effects, a classic text handler, a rearranging shuffler, and an empty shell source code builder. I have tried to keep the programs and modules general enough and simple enough that they will run on most any brand or version of Apple or Apple knockoff.

A stuffed-full and double-sided companion diskette is available with all the source and object code used in the book. Source code is provided in your choice of EDASM or S-C Assembler formats. Either way can be used as is, or else easily adapted to most any present or future assembler of your

choice. Naturally, this companion diskette is fully unlocked, easily copied, and bargain priced.

No royalty or license is needed to use any of the ripoff modules in your own commercial programs, so long as you give credit and otherwise play fair. You can order this diskette directly from me by using the order card in the back of this book. A feedback and update card is also included. An aggressive and well supported voice hotline service is provided free with your diskette order.

By the way, I'd like to do a really advanced sequel to this book that would cover such things as the new 65C02's with their literally millions of new op codes and addressing modes just waiting for your use, review some really old stuff like the Sweet Sixteen and the old floating point routines, check into Apple organization and memory maps more, look into the lie's fantastic new opportunities, do many more ripoff modules, and lots of extra stuff like this. Be sure to use the response card to tell me exactly what you want to see, and which new ripoff modules should be included.

But, getting back to here and now, don't expect this book to teach you assembly programming, because assembly programming cannot be taught. Assembly of machine language code has to be learned through great heaping bunches of hands-on experience and lots of practice. Careful study of other programs is also an absolute must. Hopefully, you can use this book as a guide to show you the way through your own learning process.

Oh yeah. It is disclaimer time again. **Apple II** is a registered trademark of some obscure outfit out in California. All of the usual names like **Atari**, **Zork**, **Scott Adam's**, **VisiCalc**, etc., are registered trademarks of whoever. Special thanks to Bob Sander-Cedarlof of **S-C Software** for his thoughtful proofing comments.

As usual, everything here is pretty much my own doing, done without Apple's knowledge or consent. Which, of course, makes it even better.

— Don Lancaster
Fall 1983

**This book is dedicated to the secret of the red wall.**
**May there always be one more.**

# 1

# What is an Assembler?

Virtually all the winning and truly great Apple II or Iie programs written today run in machine language…

**MACHINE LANGUAGE —**

**The detailed, "ones-and-zeros," gut level commands a microcomputer must have to do anvthing.**

For instance, if the Apple's data bus is presented with the binary pattern **1010 1001** and then with **1011 0111**, the 6502 microprocessor will fill its accumulator with the value hexadecimal $87, equal to decimal 183. This is done in the immediate addressing mode, as a two byte instruction.

If the previous paragraph looks like so much gibberish, you are not nearly ready to even think about reading this book.

To continue, you must know about and must have used 6502 op codes, and must completely and thoroughly understand addressing modes and hexadecimal notation. Memory maps, working registers, and address space must be second nature to you. You must also have already handwritten and hand debugged several of your own machine language programs.

Once again, this book is about assemblers, and there is NO WAY you should even think of using assemblers and assembly language until long

after you have handwritten and then hand debugged not less than several hundred lines of machine language code.

One place to pick up this machine language background is with the discovery modules in **Don Lancaster's Micro Cookbooks**, Volumes I and II.

**If you have not done all your hex homework, can't tell immediate from page zero addressing, or otherwise haven't paid your machine language dues, then please...**

**GO AWAY!**

Now that the air is cleared, and the techno-turkeys have left, let's sweep up the worst of the feathers and continue. The trouble with machine language programs, as you undoubtedly know by now, is that there is lots of tedious dogwork involved in writing them.

It is rather hard to insert something new into a hand-coded machine language program, since you'll have to move everything down on your programming form to make room for new stuff. Removing code creates the opposite problem. Even a common beginner's mistake such as the wrong addressing mode can completely mess up your program. This can easily happen if you have to substitute a 2-byte for a 3-byte instruction.

Calculating relative branches is a royal pain, particularly the forward ones where you don't quite know where you are headed. You must, of course, eat, sleep, and breathe with your 6502 pocket card before you can do any decent machine language programming. You have to know the exact address you are going to jump to, and the exact length of your code, and the exact starting point before the program will work. And you must, of course, do everything in hexadecimal, even if you really want decimal numbers or ASCII characters.

And those are just a few of the hassles. You probably have a lot more pet peeves of your own. You can automate much of the dogwork involved in machine language programming by going to an assembler…

**ASSEMBLER —**

**Any tool that simplifies or automates machine language programming.**

Most often, an assembler is a program or a program system that you run on your Apple II or IIe that helps you write machine language code. Assemblers make the writing and debugging of machine language code much easier, much faster, and much more fun. Assemblers also make it very easy to change, or edit, already existing machine language programs.

Because assembly programs are very powerful tools, there are many new skills that may be involved in learning how to use one. In exchange for this

new learning effort, an assembler will make machine language programming much faster and much more fun for you.

The tradeoff is some new effort now in exchange for lots of time saved and use convenience later. Assemblers speak a special language that is called, of all things, **assembly language**...

**ASSEMBLY LANGUAGE —**

**A "higher level" language that both an assembler and a person wanting to write machine language programs can use and understand.**

The assembler itself is really one or more machine language programs set up to interact with you as programmer and the Apple II or lie as computer. The assembler goes between you and the machine and tries to speak to the machine in machine language and to you in assembly language.

This is roughly similar to an interpreter program that can take BASIC statements understood by a programmer and convert them into machine language commands understood by the Apple II . An interpreter itself is, of course, a machine language program. So is an assembler.

Thus, you can think of an assembler as a translator that changes "people language" into "machine language." Assemblers use **mnemonics**...

**MNEMONIC —**

**A group of three or four letters that form a "word" which both the programmer and assembler uses.**

Typical mnemonics would include the command **LDX**, meaning "Please load the X register," or **ROL A**, meaning "Please rotate the contents of the accumulator to the left through the carry flag." You should, of course, be already familiar with these mnemonics for 6502 op codes.

Assembiers will often add their own new mnemonics on top of the ones already used by the 6502. An example would be the mnemonic **ORG**, telling the assembler that "Here is the original ddress where I would like you to start assembling code." More on these **pseudo-ops** later.

Mnemonics give us a shorthand way of communicating with an assembler. We could say "**1010 1001 1011 0111**" and our **6502** would know what we were talking about. But these ones and zeros sure get rough on the programmer. We could instead say, "6502, would you please immediately put the decimal value 183 into your accumulator?" This is obvious to the programmer, but the 6502's microprocessor would be very puzzled over this strange gibberish.

An assembler compromises in making use of its mnemonics that are understandable to both man and machine. The person uses **LDA #$B7** or its decimal equivalent of **LDA #183** commands to speak to the assembler,

and the assembler then recognizes and understands this to mean that the machine language coding **1010 1001 1011 0111**, or its hex equivalent of **$A9 $87**, is to go into the final program.

Another very important assembler concept is called a label…

**LABEL —**

**A name put on some value or some address or some point in a program to be assembled.**

Labels are simply names you can put on things. For instance, you could start your program with a label that says START. Other places in your program could refer to that label. For instance, to repeat your program over and over again, you could use an assembler command of JMP START as your last line. When the assembler assembles the program, it finds out where START really is and then figures out the right code to get you there. Or maybe you want a forward branch that goes to code you labeled MORE. If you don't use a label, you must know the exact address you are branching to, even if you have not gotten there yet. With proper use of labels, a good assembler will automatically figure these things out for you.

Labels also serve as memory joggers and simplify moving programs between machines. For instance, the Apple II's on-board speaker is located at **$C030**. With a label, you can define, or equate, **$C030** as **SPKR**. Every place you see SPKR in a program, you can now remember what it is and exactly what it does.

Another use of labels lets you move your program around in memory by reassembling to a different starting address. If you insert or remove code inside relative branches, those labeled branches will automatically lengthen or shorten during the assembly process. If your branch goes to an absolute address instead, the lengthened or shortened code will bomb, since the branch now goes to the wrong place.

Labels are normally five to seven characters long, and can include numbers or decimal points. Usually you have to start with a letter, and no spaces are allowed. You should try to make all labels as meaningful as possible.

There are lots of sneaky and elegant uses for labels. For instance, you can use the label "C" for a carriage return. Or labels of **G1#** and **DQ** to produce a musical note. Labels such as **MSP1** can automate linking of messages and message pointers. You can also do automatic address calculations by combining labels with the upcoming operand arithmetic.

## TYPES OF ASSEMBLERS

There are several different types of assemblers, depending on how fancy they are, what they are intended to do, and where they put their final machine language program results.

The simplest is the **miniassembler**…

**MINI ASSEMBLER- —**

**An "automated pocket card" that assembles one command at a time directly into machine language.**

A miniassembler is the smallest and simplest assembler you can get. There is a miniassembler built into the Integer BASIC code of your Apple, starting at **$F666**. The miniassembler is available either as part of the Integer ROM set, or as code booted onto a language card or into high RAM. The old miniassembler has recently been upgraded and dramatically improved as the BUGBYTER program, available as part of Apple's **Workbench** series.

The system master diskette for Apple IIe will autoboot the miniassembler code for you to power up. To activate it, type **INT**, followed by a **CALL -151**, followed by **F666G**. Or, better yet, **BRUN BUGBYTER**.

At any rate, all a miniassembler does is let you enter a mnemonic. It then converts that mnemonic to a machine language op code for you. For instance, you tell the miniassembler **0800: LOA #$B7** and the miniassembler whips out its own pocket card, and enters the code into the Apple as **0800: A9 B7**. Miniassemblers will automatically calculate relative branches for you, although you often have to make a "guess" on your forward branches.

The use rules for the Apple miniassembler appear in the BUGBYTER manual, and in the usual Apple guides and support books, so we won't repeat them. Be absolutely sure you use and understand the miniassembler and all of BUGBYTER before you try anything heavier.

A miniassembler does not allow labels and does not let you write a coded script ahead of time. You simply punch in one mnemonic at a time, and it then changes the mnemonic to an equivalent machine language op code for you. Miniassemblers do not give you a quick and easy way to "open up" code to stuff another command in, or "close up" listings to remove something no longer needed. You usually also must always work in hexadecimal with a miniassembler.

There is no really useful way to put annotation, remarks, or comments into a miniassembled program. While you are free to run your printer as you use a miniassembler, there is no really good way to get a well documented hard-copy record of what you are doing. By comments, we mean…

**COMMENTS —**

**Remarks or notes added to the instructions given an assembler.**

**Comments will be ignored by the assembler, but are most useful to people reading and using them.**

Miniassemblers also assemble code directly into the machine for you. This means that the code must go into a chosen place in your Apple where it is

expected to run. Trying to assemble code into certain areas such as ROM is futile, and trying to assemble into the stack, the text screen, or much of page zero, will bomb your Apple. So, a miniassembler is normally used to assemble code exactly where it is to run.

The BUGBYTER module is relocatable, so you can move it out of the road of your intended assembly space.

Miniassemblers are compact and very fast. They are a giant step up from writing your own machine language code by hand, and you always should learn and use a miniassembler before you go on to anything fancier…

**Before you attempt to use any fancier assembler, be sure to write and debug not less than several hundred lines of machine language code by using a miniassembler or on BUGBYTER.**

The greatest use of miniassemblers is to drive home what the assembly process is and how it works. They are also useful for quick-anddirty or very short assembly jobs. But, since there is no way to make a script of what you want to do, any later changes mean you have to miniassemble the whole job over again. Worse yet, there's no record of what you did. Our next step up leads to a full assembler…

**FULL ASSEMBLER —**

**An assembler that includes all the usual features, such as labels or comments, and the ability to work from a script that you edit and save.**

Full assemblers usually consist of a few related programs. One of these lets you write a script, or a series of instructions. You can save this script to disk, edit it, or rework it. A second part of the assembler then converts this script into actual machine language code, and gives you a printed record of the assembly process. Full assemblers use labels and make it easy to insert and remove code at any place and any time. A full assembler is normally all you need to write most machine language programs.

Full assemblers do let you put comments anywhere you like, provide for "pretty printing" for easy readability, and give you a formal printed record. There is also the macroassembler…

**MACROASSEMBLER —**

**A full assembler that also is fully programmable, letting you work with pre-defined modules, and doing other powerful tricks.**

A macroassembler will not only accept mnemonics for you. For it can also

accept a pre-defined series of instructions, and then convert all those instructions into individual mnemonics. A macro is…

**MACRO —**

**A series of instructions or mnemonics that will carry out some fancy "high level" task.**

For instance, with a full assembler, you would need a dozen mnemonics to read a text file and print one character at a time. With a macroassembler, you can design a macro that automatically will generate all the needed mnemonics for you. You would name your macro something like **PRNTX**, and just put the macro into the assembler where you wanted all the details. At the same time, you can "pass variables" through to your macro.

A really great macroassembler will even let you use the message inside the macro, such as **PRNTX /Hit any key to continue./**. Macros can be instructions inset directly into your source code, can be disk-based modules, or can be WPL routines used with "new way" editing. Which you use depends both on your choice of assembler and your programming style.

Macros are a fun tool for the advanced programmer, and they really make machine language programming fast and more understandable. But macro features are really not essential. If your regular assembler has the ability to insert routines from a disk and can do a limited amount of conditional assembly, you can "fake" many of the macro features.

It is possible to do many macro-like tasks with a supervisory word processor program, such as WPL. WPL is the word processor language that works with Apple Writer II or 1/e. We'll see WPL use examples when we get to the chapter on "new way" editing.

Some assemblers also may give you a method to separate labels that can be used anywhere in the program from labels that can only be used in one small code portion. We call these **global** or **local** labels…

**GLOBAL LABEL —**

**A label that can be used any place in the entire program.**

**Global labels can only be defined ONCE in a program.**

**LOCAL LABEL —**

**A label that can be used in several places in a program, each with a usually similar, meaning.**

**Local labels can be defined as often as needed, and will only affect a small area of the program.**

Not all Apple assemblers let you separate global and local variables, although "new" EDASM gives you a way to do this. Thus, each time you use some code module, you may have to pick a unique and different label name. Obvious ways to beat not having a local label capability are to number labels sequentially, such as **START1**, **START2**, **START3**, or to use creative misspelling, such as **PRINT**, **PRIMT**, or **PRENT**.

Yet another way to classify assemblers is whether they generate relocatable code or not...

**RELOCATABLE CODE ASSEMBLER —**

**An assembler that generates special machine language code that will reposition itself anywhere in memory automatically before its use.**

Normally, your "typical" machine language program is only allowed to sit at one exact place in memory and has only one legal starting address. This is fine, if you always know where you want your program to go.

Dino machines often speak of **virtual memory**. One of the key features of virtual memory is that any program or any program module can go at any place in memory and still work. This gets real handy when you are tacking a bunch of mix-and-match machine language modules onto the top of an Applesloth program. Virtual memory is so powerful that you can easily think of a dozen more ferinstances where it sure would be nice to put anything anywhere and still have it work. For micros and personal computers, relocatable code is a powerful idea whose time has come.

You can write machine language programs that can go anywhere in memory, so long as the code never calls itself or refers to itself with any absolute addresses. This means no absolute self-references such as loads or stores, no jumps, and no subroutines. This would be a simple example of a program that is self-relocating and can run anywhere. The disadvantage, of course, is that you aren't allowed to use most of the useful or interesting 6502 op codes when you try this.

Or, you can write a long and fancy machine language program that first finds out where it is sitting in memory, and then changes itself so it will run in its present location. The standard Apple II way of finding out where a program sits is to jump to a subroutine in the monitor with a known immediate RTS return, and then dig into the stack to find the calling address. From this point, you can play games that selfmodify the rest of the code so it works where it is sitting.

The Apple people have gone one step better, and now have "R" files. These "R" files are relocatable code modules, that work with special loader software to put a machine language module anywhere in memory. They do this by dragging along a data table that lists everything that references absolute locations. These locations are changed as needed.

If you want to use "R" files to make your machine language code relocatable, then you have to use an assembler that can handle relocatable code without choking on it.

Relocatable, or "R" files are nice for advanced programming concepts, but let's get back to some more simple mainstream stuff. Another way to classify assem biers is by where they put the machine language program theygenerate. You have a choice of inplace or disk-based assembly…

**IN-PLACE ASSEMBLER —**

**An assembler that assembles its machine language code directly into RAM memory.**

**DISK-BASED ASSEMBLER —**

**An assembler that assembles its machine language code onto a disk based file.**

An in-place assembler will directly assemble its machine language code into the RAM of your Apple II or lie. This is fast and convenient. Often, you can test your machine language program immediately.

There are several disadvantages to in-place assembly. You are limited to shorter programs, since both the assembler program and the final machine language code must fit into memory at the same time. The machine language program may have to be moved so it can run, if the intended place for the final machine language program conflicts with the code space needed for the co-resident assem bier program.

A disk-based assembler reads a disk file as an input and generates a different disk file as an output. The files can be much longer than the space available in memory, since all the assembler has to do is keep a short stash of labels and cross-references handy. Thus, you can easily write and assemble very long programs with a disk-based assembler.

There are also no limits to where the final program code sits, since this is code stashed on a disk, and not code stuffed into the machine. You can easily assemble a program that must sit in the same space the assembler does; can overwrite text screens; can work on page zero, the stack, or the keyboard buffer. Final code is ready to use without any relocation.

The bad news here is that disk drives tend to be very slow, and that you have a long song and dance to go through when you want to test your machine language program, since you may have to get out of the assembler program, and then load and run your machine language program. The "test-modify-reassemble" round trip time can be much longer.

Newer DOS speedup tricks can ease the turnaround time. Another factor that makes this long round trip time not too bad is that many programmers, including myself, are running a printer most of the time that they are assembling. This slows down an in-place assembler to where it is almost as infuriatingly slow as a disk-based assembler can be. If you are using a slow printer for quality output, there isn't that much round trip time difference

between an in-place and a disk-based assembler. A print buffer or a spooler can speed things up a whole lot for either type of assembler. Some in-place assemblers give you the option of assembling to disk, and vice versa. This can give you the best of both worlds. Here's two more terms for you…

**MODULAR ASSEMBLER —**

**An assembler where editing and assembly routines are separate, only one of which is loaded into the machine at any particular time.**

**CO-RESIDENT ASSEMBLER —**

**An assembler where editing and assembly routines can both be present in the machine at once.**

Both have advantages. Modular assembly gives you more room for your source code and possibly for in-place object code as well. The modular routines can also be longer and fancier, since they have more "elbow room" in which to work. Co-resident assembly is faster and shortens the edit-assemble-test round trip time considerably. Sometimes you might get involved with a cross assembler…

**CROSS ASSEMBLER —**

**An assembler that is displeased or otherwise unhappy with the inane garbage it is being fed.**

Uh, whoops. Computer error. Let's run that one by again…

**CROSS ASSEMBLER —**

**An assembler running on one system but generates machine language code for a different one, system, possibly even for a different microprocessor.**

If you are only using an Apple II to assemble 6502 machine language programs that are only to run on an Apple II, then you will most likely never need a cross assembler. Cross assemblers work on one machine but generate code for a different one. For instance, you could use an Apple II to generate machine language programs that are ready to run on simpler 6502 machines, such as the KIM, AIM, and SYM gang or for a 6502 controller card. This gives you all the full resources of your Apple, including disk drives, modem, printer, and such, to let you develop programs for other machines. If you work with standards of these other machines, you can directly

download programs from the Apple to the target machine. Or else use serial ports to exchange programs and data.

Other cross assemblers may work with different microprocessor families. Thus, by going to the right kind of emulator software, you can use an Apple to generate TRS-80 code, 68000 code, Macintosh routines, CRAY 1 code, or anything else you like.

Many Apple-based assemblers will provide modules to let you do cross assembly. The S-C Assembler modules in particular let you cross assemble into dozens of different microprocessor CPUs or even into dinos.

That pretty much rounds out our survey of the types of assemblers that you might find interesting or handy to use. Our main interest in the rest of this book, though, will be in using a disk-based full assembler to generate Apple II programs for Apple II or IIe use.

### How Assemblers Work

A miniassembler works as if it were an "automated pocket card." You pick a starting address, and start punching assembly language mnemonics into the machine. The miniassembler then converts these mnemonics into the correct op codes for you.

All the op codes are figured out automatically. Different address modes are entered by special symbols following the mnemonics. Relative branches are also automatically calculated, although you do have to take a guess at forward branches and then "repair" the guess when you get to the place in the program the branch is supposedly going to.

A miniassembler only works on one mnemonic at a time, and has no way of remembering what it did before or anticipating what it will do in the future. There are no labels, limited comments, and limited printed records. There is no record of what goes into the miniassembler, unless you create one yourself using some programming form or disassembly listing.

The miniassembler is an essential "go-between" step that should separate your first hand-coded machine language programs from your use of a full assembler. A miniassembler gives you insight into what the assembly process is all about, drives home the need to understand address modes, and forces you to become a better and more thoughtful programmer.

Serious programmers soon demand more than a miniassembler can deliver. So they step up to either a full assembler or a macroassembler.

Both these work pretty much the same way. All a macroassembler does is give you some more features and some extra bells and whistles to make your programming efforts more legible and more convenient.

Think about how you hand code a machine language program. First you decide what you want to do. Then you actually do the encoding process to come up with the correct op codes and addresses.

Full assemblers work the same way. First you write a script that will tell the assembler exactly what it is that you want done. Then you feed the script to the assembler, and it takes the commands in that script, and then goes ahead and tries to build a machine language program for you.

There are two stages involved in using an assembler…

**TO USE A FULL ASSEMBLER —**

**FIRST, you write a script or a series of instructions telling the assembler exactly what it is you want done.**

**SECOND, you send this series of instructions to the assembler so the assembler can use these instructions to write your machine language program.**

You go to the assembler and say "Here is what I want done." The assembler then takes this listing of what is to be done and then actually tries to do it, generating you a machine language program.

The script, or series of instructions is called the source code…

**SOURCE CODE —**

**The series of instructions you send to an assembler.**

**Source codes are written as an English text, but there are rules that must be EXACTLY followed.**

The assembler then reads your source code and tries to make some sense out of it.

you obey all the rules, the assembler will take the instructions in the source code, and follow its built-in rules so it can generate a machine language program for you.

This generated program is called the object code…

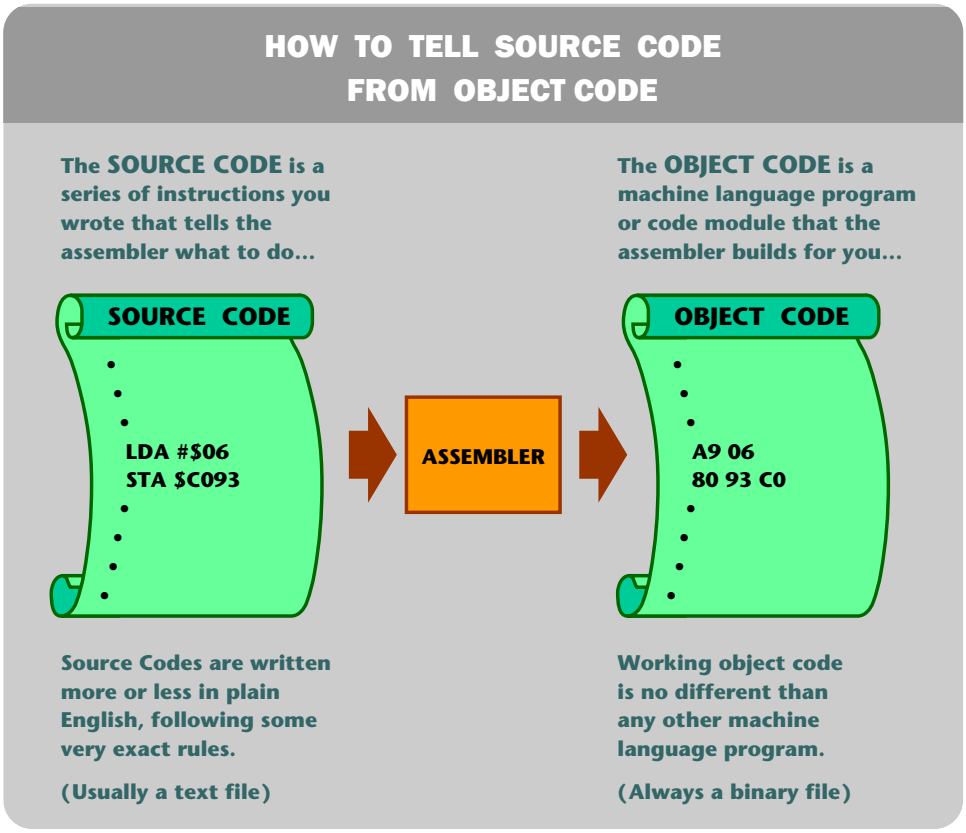
**OBJECT CODE —**

**The machine language program that the assembler produces for you, following the instructions in the source code.**

You write a script called the **source code**. The assembler then takes the source code and converts it into a machine language program called the **object code**.

Source code is sometimes called the **source file**, and object code is sometimes called the **object file**, particularly on disk-based assemblers. Either way, the source is your script and the object is the machine language result.

Like so…




Should files or tables of data also be needed, a good assembler will also produce these for you, starting with ASCII values or else a string of hex or decimal numbers.

Note that the source code and the object code are totally different types of beasties. The source code is a series of English-like instructions that you wrote. The object code is the machine language program the assembler has generated for you…

**Source code files and object code files are totally different.**

**DON'T MIX THEM UP!**

**Source code = your instructions**
**Object code= assembly result**

In the assembler we will be using in this book, the source code is usually stored on the diskette as a **text file**. The object code, of course, must be stored as a **binary file** since it is a runnable machine language program or some part of one.

Other assemblers may instead store their source code as a binary file, as a

text file, or may use some special format.But, no way will any source code run as a machine language program, ever!

So…

**OBVIOUSLY—**

**Source and object code files MUST ALWAYS have different names!**

The reason for this, of course, is that a source file is one thing (instructions from you), and an object file is something entirely different {machine language code the assembler generates). If you give these totally different code files the same name, then you'll get into the same troubles you would if you put two identical names on any pair of files on the same diskette.

There are at least three ways you can name source and object files so you can tell they belong together, yet still separately recognize them. One way is to add something to the source name to say it is indeed a source file.

A second method is to add something to the object name to say it is obviously an object file. The third route involves prefixes.

For instance, if you are working with a program called **SNARF**, you might call your source code **SNARF 1.0.SOURCE**, and your object program **SNARF 1.0**. I like this route, since your final machine language program is properly named for final use.

One other alternative is to call the source program **SNARF 1.0** and the object program SNARF 1.0OBJ0. This is the "default" way the 1.0.0BJO. assembler we will use in this book does things, so apparently someone somewhere must like this strange notation. Other assemblers might drop the version count following ".OBJ".

Others prefer to use prefixes, such as **S.SNARF** for a source code file and a **B.SNARF** for the binary object file. You should always keep track of the version of a program by adding numbers to the name…

**Add version numbers to all of your programs, always making the latest program have the highest number.**

The usual way you do this is to call your updated programs **SNARF 0.1**, **SNARF 0.2**, **SNARF 0.3**, and so on.Use tenths for small changes and routine updates. Use tens or hundreds for major overhauls. For instance, you keep adding **SNARF 0.4**, **SNARF 0.5**, and so on for any smaller changes or improvements.

Should you "refocus" your program into something wildly different, start over again with **SNARF 10.0**, **SNARF 10.1**, and so on.

**NEVER** overwrite the last working copy of anything you have…

> **NEVER overwrite the last good copy of your source code!**
>
> **ALWAYS add a new version number higher than the previous ones.**
>
> **Delete old code ONLY when it is many versions behind and cannot possibly have any more use.**

In short, **back everything up six ways from Sunday**. Don't throw anything old away till you desperately need disk room. Even then, be very careful and save a printed record. Sometimes when you think you are "improving" source code, you may actually be destroying it, or else throwing the baby out and keeping the bath water. There's also the random glitch that destroys a file. Either way, with no backup, you end up in deep trouble.

Most full assemblers will have a program that will make writing the source code script easy and fun to do. This part of an assembly system is usually called the editor. The editor is pretty much a word processor program that is set up to exactly follow the rules needed by the part of the assembly system that is to generate the machine language result for you.

The part of the assembly system that takes an edited source code script and converts it into a machine language program is called the **assembler**…

> **EDITOR —**
>
> **That part of an assembly system which helps you create or modify source code.**
>
> **ASSEMBLER —**
>
> **That part of an assembly system which takes the source code instructions and converts them into object code.**

This can get sticky fast, for "editor" can mean two different things and "assembler" can mean two different things. When you are talking about the big program, most people say "assembler" when they really mean "assembly language development system." Two important parts of most typical Apple assembly language development systems are a way to create and modify source programs, called the editor; and another separate way, called an assembler, to take the source code file and generate an object code file.

To further foul up the works, the editor part of the assembly language development system is used both to create and modify source code files. The process of using an editor to create a source code file is sometimes called **entry**, while the process of using an editor to change an existing source code file is sometimes called **editing**.

You can also do your editing and entry with a word processor, in which case you can call what you are doing anything you care to.

The context usually will help you out. An assembler is either the whole development system, or else just that part of the development system that does the actual assembly. An editor is either the whole module that lets you create or modify a source code file, or else just thatpart of the program that is actively involved in changing or correcting an existing file.

The way a full Apple assembler works is that you first use the editor part of the development system to create a source code file. As a reminder, this source code file is a series of English-like instructions. You might also use the editor to change or correct existing source code.

The assembler reads the source code file several times. Each time is called a **pass**, and most assemblers take at least two passes to complete the assembly process. The as.sembler goes all the way through the source file to find all the labels, all the definitions, and any forward branch references. It saves this data in suitable tables or lists. Then, the assembler makes a second pass to convert all these references into useful object code.

You first write source code, and then have the assembler assemble it for you. Then you test the code. Should you not like the results, you go back and change the source code to correct or improve.

The edit-assemble-test process goes round and round many times. It is not unusual to need dozens or even hundreds of cycles through the works to get what you finally want.

Most assembler programs will generate error messages for you. An error message simply means that what you sent the assembler was so stupid that it couldn't figure it out. Naturally, you can feed the assembler perfectly correct instructions that will still generate worthless or nonworking code…

**The only thing a message of**

**"SUCCESSFUL ASSEMBLY: NO ERRORS"**

**tells you is that you have not done something so incredibly stupid that the assembler couldn't figure out whatit was you were trying to tell it to do.**

**It is very easy to successfully assemble totally worthless code.**

You will definitely be seeing much more on error messages. Will you ever. The useful thing about error messages is that they will point directly to the place in the source code where any problems were found that are so bad the assembler can not, or at least should not, continue.

To recap, full assemblers and macroassemblers consist of at least two related programs. One program, called an editor, lets you create a script or source code file that explains what it is that you want done. Followed by a second program, usually called the assembler, then makes the several

passes through the source code file, and converts these instructions into a machine language code object file.

If the assembler finds any really bad problems, it will give you error messages. Some of these errors will stop the assembler dead in its tracks; others are just brought to your attention for a later repair. But, a lack of error messages in no way means that your final machine language program will work or that it will do what you want it to.

As we'll find out later, you can also use a word processor as a "new way" to create and edit source code, compared to the "old way" of using the editor in the assembler package.

By the way, on non-Apple or non-6502 machines, an "assembler" may be just that-a way to assemble programs with no provision whatsoever for entry or editing of source code. Beware of this dino trap.

### Which Assembler?

Very simply, there is no single "best" assembler for the Apple II or IIe, nor is there likely to ever be one. You find a package that suits your needs and your personal programming style, and then go with it.

Here are several of the more popular current Apple II and IIe assembly development systems…

**SOME APPLE II AND IIe ASSEMBLERS**

**ALDS**
author unknown
(Microsoft)

**Apple EDASM Assembler**
by John Arkley
(Apple Computer)

**Big Mac**
by Glen Bredon
(A.P.P.L.E)

**Edit 6502**
by Ken Leonhardi
(LJK Enterprises)

**LISA**
by Randy Hyde
(Lazer Systems)

**S-C Macro Assembler**
by Bob Sander-Cedarlof
(S-C Software)

**The Cheap Assembler**
by John Cox
(Thunder Software)

**Apple Assembly System**
by Paul Lutus
(Hayden)

**Assembly System**
author unknown
(Stellation Two)

**Boothware 8073**
author unknown
(Micro Basics)

**Merlin**
Glen Bredon
(Southwestern Data)

**ORCAM**
by Mike Westerfield
(Hayden)

**The Assembler**
by Alan Floeter
(MicroSparc)

**Tempered Assembler**
author unknown
(Avocet Systems)

We will put all the addresses and phone numbers in Appendix B to keep things orderly.

The Apple EDASM assember is really three different assembly packages. "Old" EDASM was written by Randy Wiggington and has been around for quite a while. There are two "new" EDASMs, both written by John Arkley. One "new" EDASM is DOS 3.3e based. The second one is ProDOS based. All three EDASMs are "alike but different some how." See Appendix A for a summary of the key differences. Both "new" EDASM versions are available as toolkits in Apple's **Workbench** series.

Of the others listed here, **Boothware 8073**, Avocet's **Well Tempered Assembler**, and Stellation's **Assembly System** are all specialized cross assemblers, while most of the others are general-use full or macroassemblers. **Merlin** is an enhanced commercial version of **Big Mac**.

The price of these assembly systems presently ranges from $22 to $400. Very interestingly, **the value of each of these assembly systems is almost a perfect inverse of their pricing!** Thus, the more you pay, the less you get. I guess it was bound to happen sooner or later.

For someone else's opinion of these programs, check into **Peelings** Volume 3, Number 2, February, 1982. More current reviews are also likely to appear in **Peelings** and **Infoworld** as well as in all the usual Apple magazines and review anthologies.

We are purposely not going to give you a blow-by-blow comparison of all these different assemblers. Instead, we are going to use Apple Computer's own recently upgraded and overhauled assembler for the rest of the book. This one is called The **Apple 6502 Editor/Assembler**, or **EDASM** for short, and is found on one of two popular utility disk ettes in the **Workbench** series. Both DOS 3.3e and ProDOS versions are available. These diskettes cost around $75, but since there are lots of other goodies on the disks, particularly **BUGBYTER** and **HIRES Character Generator** systems and new character fonts, your actual cost for the assembler will end up much less. Unbundled, EDASM is the cheapest assembler available.

Why this assembler? Well, first, I like it. Secondly, I use it for all my work, and it is the one I know best and have used the most. We also use it for commercial program development here at **Synergetics**, and for several of the microprocessor courses over at EAC, our local community college.

EDASM is probably the most popular assembler, if for no other reason than there are great heaping bunches of copies of the DOS Toolkit in circulation. EDASM is normally a disk-based assembler, so it can handle programs of any length, particularly very long ones that cannot easily be handled in one piece by the others. EDASM also does relocatable code assembly very well.

EDASM's recent overhaul now includes new macros, in-place assembly, optional ProDOS compatibility, co-resident assembly, along with many other new and most useful features. Important differences between "new" and "old" EDASM are summarized in Appendix A.

And, programs written under EDASM, seem to me to be much "cleaner," much easier to read, and much more well thought out and far better documented than some of the others I have seen that use competitive assemblers. This, admittedly, is a rather subjective opinion. It might just be

that more people are using EDASM, or perhaps that I may be looking in the wrong places.

Naturally, the "best" software is almost always available from sources other than Apple Computer. This goes without saying. But I have yet to find anything unacceptably bad about new EDASM. Incidentally, others consider the 5-C Assembler the "best" available, no holds barred, while Big Mac is often rated as the "best bargain."

Critics are quick to point out that EDASM has some limits to its macros and cannot easily separate global and local variables. They delight in EDASM's much slower speed and painful reloading when it is not in its in-place assembly mode.

There are also some minor peeves, such as needing an extra "A" at the end of accumulator mode addressing, inconsistency between how you exit the entry and editing modes, some overly strict page zero addressing rules, and a printer bug that sometimes messes up the top line of continued listings.

You can minimize the impact of these disadvantages. For instance, you can fake almost all of the things an in-code macro is supposed to do by building up a source code macro library on your diskette, and pulling off these modules as needed. Most of the time, many assembly language programmers will keep their printer running during an assembly. This way, you always have a printed record of exactly what you have at any time. If you do keep your printer on, the disk-based assemblers really aren't that much slower than any other, since the printer is usually holding up the works.

A spooler or a print buffer could, of course, be added to speed things up.

And, yes, EDASM's editor is dismal, dreary, and dumpy. Putrid even. But, as we'll find out later, you simply do most of your entering and editing with Apple Writer //e instead, and handle some of your macros with a glossary or else with WPL. Which instantly converts one of the worst editors into one of the most powerful available. More on this in chapter five.

Anyway, I like EDASM, and I use it a lot, and we are going to use it here.

But…

**Do not EVER buy ANY assembler program until you have had a long talk with someone who believes in and consistently uses that program!**

The main impact of new EDASM on this book will take place in the chapters that follow. Since any assembler has to do the very same things that EDASM does, you should be able to edit these chapters with margin notes any time you find differences between the details of how EDASM works and how your assembler works. We'll even give you extra room for this every now and then. All the detailed ripoff modules should work with any full or macroassembler of your choosing.

### Tools and Resources

One assembler program and one assembler book will in no way make you a decent assembly language programmer.

I have yet to see a decent Apple II assembly language programmer who was proud of the work he did last week, let alone last year. Assembly language programming is a continuous learning and skillbuilding process.

So, those who think they are going to instantly become fantastic assembly language programmers are both deluding themselves, and ripping off their customers as well. If you are unfortunate enough to ever meet one of these dudes, please go out of your way to talk him into writing programs for non-Apple machines. Send him to Honeywell. Teach him COBOL. You will kill two birds with one stone. Instead…

**The only way you can become a halfway decent machine language programmer is through lots and lots of practice and much hands-on experience.**

**The time frame involves years, and not just days, weeks, or months.**

But, as someone once said, "The longest journey starts with a single step." If you want a shot at the brass ring and want to join the club, you gotta start somewhere, sometime. Like now. That's a mighty big bag of nickels up for grabs.

Maybe some time can be saved by showing you what assembler system I use and what tools and resources I work with. One way to find out.

Here's two possible assembly language programming setups…

**ASSEMBLY LANGUAGE WORK STATIONS**

**Apple II Computer with 48K RAM**
**Integer ROM set in mainframe**
**Absolute reset ROM in mainframe**
**Applesoft ROM card with Autostart**

**-— or, preferably —-**

**Apple lie computer with 128K RAM and custom "absolute reset" EPROM monitor ROM**

**-— plus —-**

**Two disk drives**
**Quality daisywheel printer**
**Metal printwheels**

Some comments on these arrangements. First, it is absolutely essential on

older Apples that you have an "old" monitor ROM in your mainframe, if you are at all serious about assembly work. Besides the very handy single-step, trace and debug features, this old ROM lets you stop any program at any time for any reason, under absolute control. The Integer ROM set gives you the old miniassembler, the programmer's aid, and access to the "Sweet 16" pseudo 16-bit machine routines, along with the old floating point package.

The Apple lie is, of course, a much better choice for developing newer software. But you will definitely want to provide your own custom monitor EPROMs to pick up absolute reset and eliminate the obscene "hole-blasting" restart of the stock monitor chips. While you are at it, throw in a 65C02 as a new CPU, since these do so much more so much better. But…

**If you are at all serious about assembly language programming, you MUST have a way to do an absolute and unconditional reset.**

**On older Apples, this takes the "old" F8 monitor of the Integer ROM set in the mainframe.**

**On the Apple lie, this takes a pair of custom 64K E PROMs that replace the monitor ROM's**

On older Apples, you can either use a ROM card to pick up the Applesoft ROMs and the autostart ROM, or else go to a RAM card and Applesoft software. The RAM card is probably the better choice today, but should be modified for absolute hardware control. By the way, old monitor ROMs are often advertised in Computer Shopper, usually for $10 or less each.

A second disk drive is handy and almost essential. These days, you can get good drives much cheaper from sources other than Apple. I use a u-SC/ as my second drive. Sometimes you can hold off on tasks that really need two drives and borrow a second drive just long enough to get the job done.

One useful advantage in mixing your brands of disk drives is that **they all will sound different while running**. If you ever activate the wrong drive, this "aural feedback" makes it known pronto.

A dot-matrix printer can probab1y be the best choice for writing and debugging programs, because these printers are very fast. But, it is absolutely inexcusable to ever publish any dot-matrix listing, even if your uncle is an optician…

**Don't EVER publish ANYTHING that you have printed on a dot-matrix printer!**

Now, there are a few people around who claim they can actually read a dot-matrix printout, particularly from newer model printers. This peculiar genetic deficiency usually shows up in inbred generations of dot-matrix printer salesmen, but is thankfully rare otherwise.

Unfortunately, the printing processes in use today cannot read or accept dot-matrix printout. By the time your dot-matrix listing goes through a bad ribbon, gets reduced, is photocopied, gets burned onto a plate, and finally gets printed, you will end up with a royal mess.

So, I use an older Diablo 630 daisywheel printer myself, since I can't justify having one printer for listing and debugging, and a second to generate camera-ready copy.

Of course, you use a film ribbon, and for your final copies, you use single-strike film. Naturally, it is totally inexcusable to ever retype or typeset an assembly listing, because errors are certain to be added. Errors that are hard to find and harder to correct.

By the way, every now and then some turkey will try to tell you that you cannot tell the difference in print quality between a metal and plastic printwheel. This is true only if (A) you are at least a thousand feet or more away from the page, and (B) you are blind.

There is as much difference between a heavier metal printwheel and a plastic printwheel as there is between the plastic one and dotmatrix print quality. This is especially true if you use one of the "heavier" metal fonts not available in plastic, and do so on a freshly adjusted machine. I am kind of partial to the **Titan 10** metal wheel myself for listings, and to the **Bold PS** wheel for everything else.

While we are on the lookout for turkeys, watch out for misleading speed claims on newer daisywheels. The newly discovered "words per minute" rating is ten times the industry standard "characters per second" speed rating. Thus, a daisy rated at "120" is much **slower** than one rated at "40."

Even worse, the term "letter quality" has been bastardized into "near letter quality" or "correspondence quality." Well, "letter quality" means "looks like an old mangy Selectric." "Correspondence quality" means "not quite totally illegible." And, the "near" in "near letter quality" means the same thing as "nearly" getting a job, "nearly" winning a contest, or for that matter "nearly" getting run over by a garbage truck. A suitable synonym for "near" is "ain't."

Summing up, if you can, use a fast printer for assembly development and checkout, but be sure to use a good printer for your final published listings.

So much for the system. One of the really great things about the Apple II is that it forms its own superb development system. Would you believe that other computer systems make you buy program development hardware that can cost tens of thousands of dollars?

Fortunately, all you need to write good Apple II programs is a good Apple II or lie computer.

### Working Tools

What about other tools? What else do you need? I'd call a tool **anything you use or refer to while you are using an assembler to write your machine language programs**. Most of the tools that are useful are books of one kind or another. But the crucial difference between any old book and a tool book is that the tool book gets used over and over again, while any old book just sits on the shelf.

Anyway, here is a list of the tools I find handy…

**TOOLS FOR ASSEMBLY PROGRAMMING**

**6502 Pocket Card (Rockwell)**
**6502 Plastic Card (Micro Logic)**
**6502 Programming Manual (Rockwell)**
**6502 Hardware Manual (Rockwell)**

**Apple II Reference Manual (Apple)**
**Apple DOS 3.3 Manual (Apple)**
**Apple Assembler Manual (Apple)**
**Applesoft and Integer Manuals**

**Old Apple Red Book (Out of print)**
**Apple Tech Notes (IAC)**
**Apple Monitor Peeled (Dougherty)**
**What's Where in the Apple (Micro Ink)**

**Beneath Apple DOS (Quality Software)**
**Hexadecimal Chronicles (Sams)**
**Lancaster's Micro Cookbooks (Sams)**
**Enhancing Your Apple II (Sams)**

**Printer manuals and repair tools**
**Paper, ribbons, diskettes, etc.**
**Page highlighters, all colors**
**HI RES and LORES screen forms**

**A quiet workspace**

Many of these tools are obvious. Once again, addresses and phone numbers appear in Appendix B. We won't show prices or version numbers, because both are bound to change. As a disclaimer, this list is my choice and what I use. There's lots more and lots newer stuff available.

Going down this list, a **pocket card** is far and away your single most important tool. Pocket cards give you quick answers to questions like "Can I load X, absolute indexed by Y?" (yes); or "Can I do an indirect subroutine call?" (no-but you can JSR to a JMP indirect); "Can I set the V flag?" (not directly with software).

The pocket card also tells you how long the instructions take to execute. This is essential knowledge for any program that involves critical timing, and can be handy in any program.

The 6502 plastic card is equally useful. I use both. You can write on the plastic card with a grease pencil, but you can't fold it and carry it with you.

The 6502 Software Manual is also indispensable. You simply cannot do any assembler work without this book. The book was first written by MOS Technology, Inc., and for its time was one of the finest technical manuals ever produced by any semiconductor house. It is a classic in every sense of the word.

Those **MOS Technology Inc.** blue originals in their rugged dark blue covers were big, sturdy, and quite easy to read. Most of today's knockoffs

by Rockwell International and Synertek are smaller, have lower print quality, and are harder to read.

The **6502 Hardware Manual** in the same series isn't nearly as well written or understandable, but it is also an important tool for the assembler programmer.

You will want all the usual Apple manuals, particularly the lie technical reference manual and the DOS manuals. The Applesoft and Integer manuals and tutorials will be handy if you are tying machine language modules into BASIC, rather than writing decent all-machine programs.

You will also want to get access to a copy of the **Apple Tech Notes**. This thick series answers many Apple-use questions and spells out all known Apple bugs to date. All International Apple Core (IAC) member clubs have a copy. Or, if you can find a reasonable Apple dealer, they might let you look at their copy. You can buy these tech notes, but they are expensive.

What you won't be able to buy is a copy of the **Apple Red Book**. This was the original Apple II system manual. Among its priceless goodies is a schematic that is orderly and function-oriented rather than the intentionally confusing mess shown in the pre-lie manuals. You'll also find complete details on the Sweet 16 sixteen-bit software commands, detailed miniassembler listings, the original tone subs, low-cost serial interfaces, and listings on the original floating point package. Very handy and very essential if you are still working with an Apple II or II+. You'll have to copy this one on your own, as the Red Book is definitely out of print.

Note that the Apple lie technical reference manual does not come with a IIe and has to be ordered separately at extra cost. This manual is absolutely essential for lie assembly language programming.

The **Apple Monitor Peeled** is a very dated book. But, I still find it useful to understand and use the monitor features, while the "must have" **Beneath Apple DOS** gives you one thorough treatment to the disk operating system. **What's Where in the Apple II** is a detailed addressby-address listing of all known major uses of all memory locations in the entire machine. There are two parts to the listing. One part is arranged numerically and the other alphabetically. These listings are an update and extension of the original that appeared in the August 1979 issue of **Micro**. I use both the original article and the new book, because the original is easier to use and only takes a few notebook pages.

The **Hexadecimal Chronicles** (Sams 21802) is a reference that instantly gets you from decimal to hex to Integer BASIC's inverted decimal, and back again, along with ASCII conversions, and includes a hex arithmetic and circular branch calculator, and bunches of other goodies. This one is most useful when you are tying machine language subs to BASIC programs, or are moving BASIC pointers around to protect or capture a machine listing.

Volumes I and II of my **Micro Cookbooks** (Sams 21828 and 21829) should be a good way to pick up the fundamentals of hand-coded machine language programming. This is done through a series of discovery modules that lead you through most op codes of the 6502. You must use these discovery modules or something similar before you can even think about working with assemblers.

The **Enhancing Your Apple II series** (Sams 21822, etc ... ) can provide

you with many examples of machine language program modules and use ideas. In particular, the "tearing" method in Enhancement 3 of Volume I is essential for any assembly language programmer, since it shows you an astonishingly fast way to tear apart and understand unknown code.

You will also want a complete set of maintenance manuals and repair tools for your printer. These are usually not provided with your printer purchase. Note that most printer people charge bunches extra for their real service manuals. Often these will be broken up into a spec manual, a repair manual, a spare parts list, a price list, and special tools.

There are many other programming aids, support books, utility diskettes, and so on that are heavily advertised. I find myself buying but never getting around to using these. Around 90 percent of what's available is less than useless, so always check with someone that believes before you buy.

Naturally, you'll need some diskettes, tractor paper, film ribbons, and all the usual stuff like this. A complete set of page high lighters are also essential to have on hand. These are very useful for identifying changes and corrections on printouts and are absolutely essential for the "tearing" disassembly method to work.

### What About Machine Programming Books?

You'll find dozens and dozens of books around that claim to teach you all you will ever want to know about 6502 machine language or 6502 assembly language programming, and then some.

Usually, you buy these books by the running yard, with a price of $28 per inch or so being typical. Mill ends are slightly cheaper. Put them on your bookshelf to astound your friends. Or, if you happen to have a table with a missing leg, put a stack of them to their only known use.

Very few people ever read these books. In fact, most of these books have been designed from the ground up so you could not possibly read and understand them even if you wanted to.

A very select few of these books are genuinely outstanding. Unfortunately, most of the others are utterly atrocious ripoffs. The trash-togood ratio here is well over 30:1 and is steadily and appallingly climbing.

And even if everybody else thinks some title is a great book, it may not suit you, since its level may be too advanced, or too simple, or locked into some obscure trainer, or too pro-dino, or too far off in left field.

So, let us repeat what we said earlier about assembler programs, only this time we'll apply it to programming and assembly books…

**The overwhelming majority of all programming and assembly books will NOT meet your personal needs.**

**Do not EVER buy ANY assembler book or any machine language book until you have had a long talk with someone who believes in and uses that text!**

The safest thing to do is to wade into the lair of your nearest Apple machine language freak and find out which books are out front, have torn or missing covers, and are thoroughly thumbed over. Don't even consider a book that has nothing spilled on its pages.

Come to think of it, though, it is never safe to wade into the lair of your nearest Apple machine language freak. Forewarned is forsooth, though, or whatever.

At any rate, **avoid buying these books unless you do happen to want a complete set of "one of each."** But that gets expensive in a hurry.

## Software

The software you use will, of course, depend on which assembler you chose, and what else you decide to have on hand. Here is what I usually work with…

**SOFTWARE FOR ASSEMBLY PROGRAMMING**

**EDASM and BUGBYTER**
**System Master**
**DOS Toolkit**
**Inspector**

**Bag of Tricks**
**Apple Writer IIe**
**Copy II Plus**
**Enhancing Diskettes**

**EDASM** is the Apple assembly development system. Be sure to use one of the greatly improved "new" versions, either for DOS 3.3e or ProDOS. You should be familiar with the System Master diskette by now, particularly the program FIO. The DOS Toolkit holds EDASM and BUGBYTER, along with several other interesting utilities and HIRES character fonts. **Inspector** is one of many available file utilities, while **Copy II Plus** is a versatile and informative copy and disk speed program. **Locksmith** is comparable.

**Bag of Tricks** is most useful for fixing bad diskettes. **Apple Writer IIe**, of course, is a great word processor and should get used for all of your documentation, besides being a better editor than the one in EDASM. The Enhancing diskettes are from the **Enhancing Your Apple II** series.

There are a lot of new utility programs available today that do things like single-step whole programs, manipulate and search files, dump ASCII strings, disassemble listings, generate cross references, provide HIRES utilities, offer ampersand links, edit diskettes, and so on.

I have not gotten around to trying very many of these. The obvious advantages of these new utilities are that they make writing and testing programs much quicker and easier. Two obvious disadvantages are that costs pile up at $30 and $100 per diskette, and that some of these programs may conflict with where you want to be in memory. A few of these are excellent; many others are less than useless.

### Assembly Magazines

Magazines are some of the best places to learn about machine language and assembly language programming. Here's my choice of the best…

**ASSEMBLY LANGUAGE MAGAZINES**

**Call A.P.P.L.E.**
**Apple Assembly Lines**
**Apple Orchard**
**Peelings**
**Hardcore**
**InCider**
**Cider Press**
**Nibble**

The finest Apple assembly language magazine is **Call A.P.P.L.E.** A complete set of these, their user library diskettes, and their publications is absolutely essential to serious machine language or assembly language programming. Their thick **All about DOS**, **All about Applesoft**, and **All About Applewriter** manuals are particularly valuable.

**Apple Assembly Line** is a funky little newsletter with fantastic vibes. It centers on the S-C family of fine assemblers, but is otherwise most readable. **Apple Orchard** is the IAC publication, and often has interesting reprints from the various newsletters. **Peelings** is the only source of well thoughtout and largely unbiased Apple software reviews. **Hardcore** has some very interesting stuff in it, but it could be so much more than it is. You'll also find very old issues of **Micro** to be most handy and informative, but this one clearly has peaked, so it is not on the list. **Cider Press** is a newsletter of the San Francisco Apple Core.

Note that this listing of magazines specifically involves Apple assembly language programming. There are other great micro magazines, such as **Byte**, **Infoworld**, **Computer Shopper, Creative Computing**, **Dr. Dobbs Journal**, **Microcomputing**,**Softalk**, and dozens more. These have all sorts of useful things in them, but they do not consistently center on Apple II or IIe machine language programs and assembly language programming techniques. There are also hundreds of club newsletters out there, many of which will have new and useful assembly tidbits.

Unfortunately, the price of the club newsletters is high, and their quality is steadily dropping. The reason for this drop in quality is that most Apple clubs are now of, by, and for users, rather than hackers. This trend is intrinsically evil and despicable. Also sad.

There's also a musical chairs game going on where everybody reprints everybody else without anything new ever being generated. Which waters down the stock something awful.

Since the trend seems away from hackers, the older issues of most club newsletters will most often have the better and more useful goodies in them, so it pays to dig back. Way back. The **Denver Apple Pi** group maintains an

on-line data base of most everything ever written in any Apple pub.

### Reprints and Anthologies

Two other essential resources are reprints and anthologies…

**IMPORTANT ASSEMBLER REPRINTS**

**Abacus Plus (ABACUS)**
**Best of Cider Press (SFAC)**
**Inside Washington Apple Pi (WAP)**
**Peeking at Call Apple (A.P.P.L.E.)**
**Wozpack (A.P.P.L.E.)**

These reprints usually show off "the best of" some year's newsletter or magazine output. They are a good way to get everything at once fairly cheaply. The argument that you are buying old information is offset by the fact that older Apple information is often better and more recent information these days. Some errors are likely to be corrected as well.

We must also mention the **Apple Avocation Alliance** as well. These people stock just about every available public domain Apple program, and will copy them for you at a cost around twelve cents each. AAA also has terrific diskette prices. Unfortunately, except for several absolute gems, very few public domain programs are worth twelve cents apiece. Nonetheless, studying these may prevent you from reinventing the wheel and should clearly and concisely show you how not to do things.

Other public domain program libraries include the extensive ones offered by the **IAC**, by **Call A.P.P.L.E.**, and the **San Francisco Apple Core**.

That's sure a long list of resources for assembly language programming. But, if you think that's bad, you should see all the garbage I bought and did not tell you about. Hopefully, these resource listings will cut down the totally ridiculous costs of getting into assembler work.

Naturally, **don't buy anything you haven't looked at first**. Work with club groups. Check into schools. Ask friends. Visit company and technical libraries. Pick up whatever works for you.

But don't try to write Apple machine language programs in a vacuum. That may have worked in 1977, b no more. Those days are long gone. Get and stay informed.

### Disassemblers

You mean that after you go to all the trouble to assemble a program, that you may want to take it all apart again? You better believe it.

The opposite of assembly is called **disassembly**. You disassemble a program or listing when you want to find out what the code is trying to do or how it is supposed to work…

**DISASSEMBLER —**

**Any tool that lets you take apart a machine language program to see what is in it or how it works.**

Naturally, it is totally inexcusable to ever buy or use any piece of Apple software without completely tearing apart the program to see what is inside and how it works. Also, naturally, the first thing you do to any locked program is make yourself several unlocked copies under standard DOS. Then generate your own modifiable assembler source code, and a complete set of working source files. You should do this automatically the first time you boot any new disk. Every time. Without fail. Far and away the best way to learn assembler programming techniques is to diligently study how others do it»

**Far and away the best way to pick up assembly language skills and new use ideas is to...**

## TEAR APART EXISTING PROGRAMS

There are several good ways to disassemble existing code. Here are the three I normally use, in order of increasing complexity…

### METHODS TO DISASSEMBLE CODE

1. **Use the disassembler in the Apple system monitor or BUGBYTER.**
2. **Use the "tearing" method from the Enhance series.**
3. **Use a capturing disassembler, such as Rak-Ware's DISASM.**

There is a "**L**" or List command in the Apple system monitor that will disassemble any program for you twenty lines at a time. For more lines, you use more L's. This disassembling lister converts object code into assembly mnemonics and shows such things as the address modes and the absolute addresses that relative branches go to. There are no labels or comments. For a printed record, you simply turn the printer on before you list the lines you want disassembled.

The "tearing" method appears in Enhancement 3 of Volume I of the **Enhancing Your Apple II** series (Sams 21822) gives you an astonishingly fast and easy way o tear apart any unknown machine language listing and provides for full comments and accurate labels.

A capturing disassembler tears apart object code and then converts it into a source code file th EDASM or another assembler can use. It puts labels on everything n eded, but these labels are simply coded sequentially. You then have to go through the listing and add your own comments and make all the labels more meaningful. Sometimes, you can predefine useful label names. A capturing disassembler usually includes a complete cross reference of who refers to whom when.

The DISASM program by Rak-Ware is the only one of these I have worked with so far. Similar products are available from Decision Systems and Anthro Digital.

DISASM does what they say it will and is reasonably priced. Their cross reference generator is particularly useful. An alternative to a disassembler is to rekey the entire results of the "tearing" method. Which is the better route depends on your programming style and the length of the program under attack. Using "new way" editing does simplify and speed up the repairs to a captured listing.

DISASM's triple cross references are most useful, though. You get internal, external, and page zero reference tables that are absolutely essential to tearing apart any major listing. Very nice.

Regardless of which disassembly method you use, there is one big gotcha you must watch for…

**A disassembler will only give you useful results if it is working on VALID code, and then only when begun at a LEGAL starting point.**

**Otherwise you get garbage.**

What this says is that **you can only disassemble code that has gotten previously assembled**. Try to disassemble a file or some data values, and you may get bunches of question marks and totally absurd op codes.

Even with legal and working code, you also have to start at the right place. If you have a three-byte instruction, you must start on the first byte. Start on the second or third byte and you get wildly wrong results. Having the wrong starting point in legal code isn't nearly as bad as trying to disassemble a data file or a text file, since the disassembler will probably straighten itself up and fly right after a few wrong listings. But watch this detail very carefully.

If you try to disassemble, say, an ASCII file instead of legal op codes, your cross references will end up giving you bunches of illegal and nonexistent "artifacts," caused by reading pairs of ASCII characters as addresses.

For instance, an "AB" ASCII pair may generate a false address of $4241, and so on. You will also get cross reference artifacts generated if there are short stashes or other files buried inside your legal op codes. These artifacts can be eliminated one at a time by hand, or by telling the disassembler to "skip over" one or more tables.

You will need both assembiers and disassembiers to do a decent programming job. One puts together, the other takes apart.

### What an Assembler Will Not Do

A car is one possible way to go to the bakery, get a loaf of bread, and then return. But there is no way a car can do this **by itself**.

You have to drive the car.

In the same way, an assembler is a great tool to help you write machine language programs, making the process easy, fun, powerful, fast, and convenient. But there is no way that an assembler will automatically write programs for you.

You have to tell an assembler exactly what it is you want done, exactly when you want it done, and exactly how to do it...

> **An assembler will NOT write machine language programs for you!**
>
> **You must tell the assembler ahead of time exactly what it is you want done, when you want it done, and how it is to be done.**

Thus, **an assembler is nothing more than a very powerful tool that will do exactly what you tell it to**. To use an assembler, you must already be a competent and knowledgeable machined language programmer.

To get into this game and go for the brass ring, you must start by hand coding and hand debugging a few hundred lines of machine language code on your own. Then you should get with a miniassembler and practice with it, again for several hundred more lines.

Next, you should use the "tearing" method to take apart and study at least a dozen major winning Apple programs. This shows you how the "big boys" do it. Finally, if and only if you thoroughly understand what machine language is all about, you should move up to a full assembler or a macroassembler.

Any other way isn't even wrong.

# 2

# Source Code Details

If you are going to have an assembler or some assembly language development system create a machine language program for you, somehow you have to give the assembler some instructions.

Once again, there is no way an assembler will write a program for you. All an assembler can do is take the exact instructions you give it and then begin from there to try and come up with some useful code.

We have seen that these exact instructions are called the source code…

**SOURCE CODE —**

**The series of instructions you send to an assembler so it can assemble a program for you.**

You can think of the source code, or source code file, as a script or a series of instructions. In this script, you will usually find op codes and "how?" or "with what?" qualifiers that go with the op codes as needed for certain address modes.

Instructions to create subroutines and data files may also be included. You most likely will also find special instructions that vaguely resemble op codes that are intended for use by the assembler, rather than becoming part of the final machine language program. We'll find out later that these are known as **pseudo-ops**.

In the script, you will also find definitions of labels and values. There will also be lots of comments/ or user documentation. Comments can include such things as a title block, the copyright notice and author credit, a description of what the program does, instructions on how to run the program, and listings of any gotchas or any modifications that might be needed. Parts of the script will also be involved in the pretty printing that makes the entire script easy to read and easy to use. Examples of pretty printing are blank lines, page breaks, and centering spaces.

To sum up, **a script or source code file contains all the information needed for the assembler to put together a useful machine language program for you**, along with all the documentation needed to tell people what is happening in the process.

In this chapter, we will find out just what source code is and how to use the "work unit" of the source code file, which is called a **line**. After we pick up these internal details, we'll go on in chapter three to find one possible way to organize and structure your source code.

Then, with this background, we'll go on to chapters four and five. Here, we'll see how you actually go about writing and then editing, or changing, a source code file for the assembly language program of your choice. Chapter four will show us the "old way" of using an editor in its intended way, while chapter five will give us full details of the "new way" of using a word processor instead. The foremost use rule involving source code is…

**The source code file is more or less a series of instructions in plain, old English, except……**

**ALL RULES MUST BE EXACTLY FOLLOWED!**

There are some very exacting and very nit-picking rules as to what goes into the script. Disobey these rules, and the assembler will generate garbage for you or simply will not work at all. In particular…

**Simple things like a missing or an extra space or a forgotten "$" for hex symbol can make the entire source code totally worthless!**

Source codes are most useful, handy, and informative. But, you absolutely **must** follow the exact use rules involved with source code files if you are ever going to get anything usable out the other end of the pipe.

## Source Code File Formats

The source code file will hold enough information to do the job you want done. The length of your source code can be just a few characters you might need for a simple patch, through part of a page for a minor subroutine,

or many dozens of pages for an elaborate or very long, full-blown machine language program. The source code will do what you want it to. You make it as long as you need to handle the task at hand.

Some assemblers put a limit on how long the source code can be. If this happens, you break the source code into logical chunks and process one chunk at a time. Then you take the machine language modules you get from this process, and recombine them into a single, long program.

The EDASM assembler we will use as our "baseline" assembler is usually disk based, and lets you write very long programs in one piece if you want to. Often, though, it is best to work in small and separate modules of your source code, combining them later.

We call the "work unit" of the source code a **line**...

**LINE —**

**The "work unit" of a source code file.**

**Eighty or fewer ASCII characters ending with a carriage return.**

**Enough information to assemble one op code; or pass a single command to the assembler; or supply a short comment or part of a long one.**

At one time, everyone in the dino computer world knew what a line was, since all messages and all communications were line oriented. Should you want to, say, process words, you had to keep each line of characters separate and work with each line individually. Now, this seems incredibly dumb, but that's the way things were. It took the micro people with their memory-mapped video to first see the completely obvious.

But there are a very few jobs remaining where it is a good idea to keep every entry on a separate and unique line that has to stand on its own and has no particular long term relation to the line above or the line below. Assembler source code files are one place where working line by line still is a pretty fair way of doing things. Quaint but fair.

If you decide to use a "new way" word processor, you will pick up "free form" or full-screen entry where you can see lots of lines at once and easily edit across line boundaries. But, you will still have to keep line oriented.

The lines in the source code are often sequentially numbered...

**Each source code line will usually be numbered in decimal.**

**The numbers normally start with one and count "by ones," in sequential order, up to "N."**

**"N" is the number of lines you need to complete the job that the source code is trying to do.**

The reason for this numbering is that we need a way to talk about or work with a single line. Instead of saying "the line with the LOA #$56 command," or "the line just above the mustard stain," we say "line number 145." Since the lines are all numbered, we can find line number 145 and work with it. More importantly, so can the assembler.

Actually, your line numbers do not normally go inside your source code. It is kinda dumb to waste disk space on things that are easily calculated and not particularly permanent. Instead, line numbers are an artifact of the editor or assembler you use. This convenient artifact is normally generated for you by counting the source code lines as they come off the disk or out of RAM and then numbering them on the way to the screen or a printer.

This type of line numbering is very obvious, but it may be different from other computer numbering schemes that you might be familiar with. As some counter examples, machine language programs are located by addresses, and do not use line numbers. BASIC programs use line numbers, but you usually skip around, counting by tens or whatever, and those lines do not have to be entered in sequential order. Pascal does not use line numbers. Instead, the relative position of the line in the program conveys what the line is and what it does.

But, none of these are assemblers. Assemblers normally have line numbers ranging from one to N, in order, with nothing missing and no duplicates. EDASM uses this "one-to-N scheme. Other assemblers might start their numbering with 1000 or 10000 to keep the number of printed digits constant. These other assemblers sometimes count by tens instead of ones.

One confusing thing about source code file line numbers is that they don’t stay unique…

**What was line 145 in one version of a source code might become line 137, or line 193, or might be just plain missing, in a later version of the same program.**

**As the program length changes, or as corrections are made, each line number may point to a different source code line.**

So, all versions of all source codes are usually numbered sequentially from one to N, counting up "by ones." No missing line numbers are allowed, nor are you allowed to put any line numbers in the wrong order. If you make the source code shorter by deleting something, all the line numbers higher than the deletion decrease in value. If you make the source code longer by adding something in the middle, then all the line numbers above the addition increase by the amount needed.

Once again, there really are no line numbers in most source code. The line numbers are an artifact generated by the editor or the assembler for your convenience. Line numbers are calculated by counting carriage returns on the end of source code lines as they come off the disk or out of memory.

Thus…

> **Regardless of the code version or the meaning of any particular line…**
>
> **Most versions of most source codes are numbered from 1 to N with nothing skipped and nothing out of ourder.**

The line numbering is usually fully automatic and is done free for you by the editor or assembler. All you have to do is make sure you really mean "line 143" when you say "line 143," because any change in the source code may change the line number.

Two nasty examples: Say you write a source code and then tell the editor to delete line six, then line eight, and then line ten. What you really did was delete lines number six, nine, and twelve, because the first deletion bumped everything above line six down a line, and the second one bumped everything above line nine down yet another line.

Or, say you get lazy or in a hurry and don't do a printer dump of each and every version of your source code as you go along. Say further that you add some innocuous line such as some extra white space some place inside your next-to-latest source code version. Now you decide the carry needs cleared. You shove the CLC line in, but what happens? Instead of being where you thought you were, you are one line off, inserting the carry one place beyond where you expected it to go.

More details on this later. One sneaky way to minimize line numbering problems is to **always edit from the high numbers down**, rather than from the low numbers up. That way you are finished with the line numbers that are going to change before they do in fact change. For now…

> **You must keep EXACT track of the line numbers by yourself!**
>
> **Line numbers may become wrong if you add or remove lines from your source code, or if you are trying to use an out-of-date printout.**

Later in chapter five, we'll see an automatic line number changer using WPL that adds, removes, or updates line numbers from word processed source code. Other assemblers may have different numbering rules or use options.

Always check.

OK. So what goes on a line? We already know that **a line is the work unit of a source code file**, and that a line is some number of characters that will fit neatly across a page or screen that ends with a carriage return.

There are different tasks that each part of a line is intended to do. These task areas are called **fields**…

**FIELD —**

**A part of a source code line that has an intended use.**

We already know about the line number field. Line numbers usually are in sequential order from one to N, with nobody missing, nobody out of order, and nobody duplicated. We also know that the line number field is handled more or less automatically for us. Just be sure that the line number you say is the line number you really mean.

We obviously need more fields. In EDASM, the others are called the **label** field, the **opcode** field, the **operand** field, and finally, the **comment** field…

**EDASM usually uses four fields in addition to the "free" line number field.**

**These are the label field, opcode field operand field, and the comment field.**

Here's a picture that says the same thing…

**FIELDS INSIDE A SOURCE CODE LINE**


The **NUMBER** field, if used, sequentially numbers each line from 1 to N. This one is automatic, so you don't worry too much about it.

The **OPCODE** field holds either a 6502 mnemonic or else a pseudo-op instruction to be used by the assembler.

The **COMMENT** field holds remarks useful to people but ignored by the assembler. You can also have source code lines that are **entirely** comments.

The **LABEL** field puts a name on the line so that it can be found or referred to. labels show jump and branch destinations and and can identify files or entry points.

The **OPERAND** field qualifies the opcode, answering "how", "where" or "with what". Operancs may be numbers, labels, addresses, or data.

The **SPACE** can be used **between** fields as a tab character. But spaces are forbidden **inside** of any and all fields.

A **CARRIAGE RETURN** ends the source code line.

More details on what each field does shortly. But, if each field is to do some unique job,we need a way to get between fields and we need a way to tell which field is where. EDASM uses the "space" character for tabbing…

**EDASM traditionaly uses the spacebar to "tab" between fields.**

**Everything up to the first space goes into the label field.**

**Everything between the first and second spaces goes into the op code field.**

**Everything between the second and third spaces goes into the operand field.**

**Everything beyond the third space goes into the comment field.**

**Because the spacebar is used to shift between fields, spaces are not allowed in any of the first three fields**. You must use spaces between fields. You must not allow spaces inside any of the first three fields…

**You MUST NOT use any spaces inside the label, op code, or operand fields!**

**You MUST use a space any time you want to go on to the next field!**

After you get to the final, or comment field, you can use any number of spaces any way you want to. But, spaces are strictly a no-no in the label, op code, or operand fields.

You might logically ask, "Why not use a tab command to tab, instead of space?" Well, tabbing was tricky on older Apples, and faking the tabs by padding spaces gobbles up space on disk or in RAM. Fur:›ther, allowing spaces in labels or op codes would create all sorts of other problems. Besides, even on a lie, the spacebar is much larger and easier to find and use than the tab key.

On "new" EDASM, you can use the lie tab key to tab if you want to. In fact, this eases "new way" editing by quite a bit. Note that using the tab command to tab is awkward on pre-IIe Apples. So, the older EDASM rule is that, after a carriage return, **the first three spacebar hits force tabs**. After that, spaces get used as spaces. Not all fields are needed on every line. But…

**Each source code line MUST have something in the op code field.**

**"Something" is either a command for the assembler or else a real op code for the computer.**

# Assembly Cookbook for the Apple™ II/IIe

## ( part one )

**Your complete guide to using assembly language for writing your own top notch personal or commercial programs for the Apple II and IIe.**

- **Tells you what an assembler is, discusses the popular assemblers available today, and details the essential tools for assembly language programming.**
- **Covers source code details such as lines, fields, labels, op codes, operands, structure, and comments-just what these are and how they are used.**
- **Shows you the "new way" to do your source code entry and editing and to instantly upgrade your editor/assembler into a super-powerful one.**
- **Shows you how to actually assemble source code into working object code. Checks into error messages and debugging techniques.**
- **Includes nine ready to go, open ripoff modules that show you examples of some of the really essential stuff involved in Apple programming. These modules will run on most any brand or version of Apple or Apple clone, and they can be easily adapted to your own uses.**

**This cookbook is for those who want to build up their assembly programming skills to a more challenging level and to learn to write profitable and truly great Apple II or lie machine language programs.**

SYNERGETICS SP PRESS
3860 West First Street, Thatcher, AZ 85552 USA
(928) 428-4073 http://www.tinaja.com
ISBN: 978-1-882193-16-5